Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра системного программирования ИСП РАН
Лаборатория (laboratory name)

Выпускная квалификационная работа бакалавра

Исследование и разработка методов машинного обучения

Автор:
Студент 082 группы
Иванов Иван Иванович

Научный руководитель:
научная степень
Денисов Денис Денисович

Научный консультант:
научная степень
Сергеев Сергей Сергеевич

Москва 2025

Аннотация

Исследование и разработка методов машинного обучения
Иванов Иван Иванович

Краткое описание задачи и основных результатов, мотивирующее прочитать весь текст.

Abstract

Research and development of machine learning methods

Содержание

1 Введение

Типовая система языка программирования служит фундаментом для формального определения его спецификации, обеспечивает статический анализ программ, поддержку оптимизаций кода и способствует улучшению его структурированности и читаемости. Высокоуровневые языки программирования(Java, C#, Kotlin) предоставляют разработчику мощные абстракции и безопасность кода посредством автоматического управления памятью и объектно-ориентированных парадигм. Многие из этих языков сохранили в себе исторически сложившееся разделение типов на "примитивные"типы и “объектные” или “ссылочные”, призванное потенциально улучшить производительность в некоторых случаях. Тем не менее, в то же время оно приводит к несогласованности в системе типов и усложняет достижение полностью единообразной и интуитивной модели программирования. Примитивные типы часто лишены полезных свойств объектов, таких как наследование, полиморфизм, вызов методов без явного упаковывания/распаковывания и использование в качестве аргумента для шаблонных типов.

В данной работе будут теоретически рассмотрены последствия устранения явной разницы между примитивными и объектными типами из высокоуровневого управляемого языка программирования и описаны этапы и результаты реализации.

1.1 Типовая система

Типовая система — это формальная синтаксическая и семантическая структура, сопоставляющая типы и программные конструкции (выражения, переменные, функции) для обеспечения чётко определённого вычислительного поведения. Её основные теоретические и практические назначения:

1) **Корректность поведения программы**

   С введением типовой системы становится возможной статическая верификация соблюдения ограничений типов, не допускающая к выполнению неверно типизированные программы. Данный механизм гарантирует:

   - *Безопасность выполнения* — предотвращение неопределённого поведения за счёт блокировки запрещённых операций (например, некорректные обращения к памяти в управляемых средах типа JVM/CLR)
   - *Теоретическую обоснованность* — соответствие двум фундаментальным принципам типобезопасности по Райту-Феллейзену:
     1. **Прогресс**: корректно типизированная программа не может застрять в промежуточном состоянии
     2. **Сохранение**: типы остаются согласованными на всех этапах вычисления

2) **Уровень абстракции кода**

Типовая система обеспечивает инкапсуляцию доменных инвариантов с помощью абстракций типов (абстрактные типы данных и интерфейсы) и позволяет формально специфицировать границы модулей.

3) **Возможность оптимизаций**

Наличие статической типизации позволяет применять оптимизации на этапе компиляции (например, специализация обобщённых типов и статическое разрешение методов) и минимизировать время исполнения в управляемых средах за счёт уменьшения количества динамических проверок типов.

1.2 Статическая и динамическая типизации

1. **Статическая типизация** (Java, C#, Kotlin):

   - Валидность типа доказывается на этапе компиляции с помощью формальных суждений о принадлежности выражения к типу в данном контексте ($\Gamma \vdash e : \tau$)
   - Гарантирует типобезопасность по Райту-Феллейзену
   - Критично для управляемых языков программирования из-за возможности проверок безопасности памяти (например, верификатор байт-кода JVM) и эффективности Just-In-Time компиляции за счёт встраивания методов, ориентированных на конкретный тип

2. **Динамическая типизация** (Python, JavaScript):

   - Типы функций и выражений определяются во время исполнения
   - Нет формальной гарантии типобезопасности, но есть возможность менять поведение объектов, модулей и классов во время исполнения

Современные управляемые языки высокого уровня (Java, C#) преимущественно основаны на статической типизации, расширенной возможностями динамического анализа во время выполнения. Данный компромиссный подход позволяет сочетать строгость формальной верификации с практической гибкостью разработки, но противоречие между статическими гарантиями и динамической адаптируемостью продолжает оставаться предметом активных исследований.

Исторически и архитектурно, многие языки разделяют типы на две категории: **Примитивные типы** (далее – примитивные типы или примитивы) Представляют собой базовые значения (например, целые числа, числа с плавающей точкой, булевы значения, символы). **Ссылочные типы** (далее - ссылочные типы, объектные типы)Представляют собой экземпляры классов, которые хранятся в куче (heap) и управляются сборщиком мусора. Они наследуются от базового класса и обладают методами, полями и другими объектно-ориентированными свойствами.

1.3 Примитивные типы

Примитивные типы являются наиболее фундаментальным типом данных в языках программирования. В отличие от объектных типов, они:

- **Не ссылают на память**, а непосредственно являются значениями. У них отсутствуют:
  - заголовок объекта (object header)
  - таблица виртуальных методов (vtable)
  - ссылочная идентичность (identity)
- Имеют **фиксированную семантику**, определяющуюся стандартами языка (например, Java JLS §4.2, C# ECMA-334 §8.3)
- **Не могут быть подтипами** или наследоваться и являются атомарными (базовыми) элементами в иерархии типов

1.3.1 Роль в конструировании типов

В теории типов примитивы являются простейшими случаями конструирования типа и не разлагаются на более простые типы. Конструирование сложных типов происходит на основе:

- **Произведения** (кортежи, записи) – (Int, Bool)
- **Суммы** (типы-суммы, перечисления) – `data Maybe a = Nothing | Just a`
- **Функции** – Int → Bool
- **Рекурсивные типы** – `data List a = Nil | Cons a (List a)`

1.3.2 Основные характеристики

- **Представление в памяти**

  Аллокация на стеке* (default for local variables in methods)

- **Производительность на уровне процессора**

  1. Арифметические операции с примитивами компилируются непосредственно в нативные инструкции:

```
; x86 assembly for 'a + b'
mov eax, [a]
add eax, [b]

```

  2. **Локализация в кэше**

     Линейная модель доступа к памяти (критично для больших вычислений)

1.3.3 Примеры в высокоуровневых языках – NEED TO BE FIXED

Таблица 1: Примитивные типы в различных языках программирования

Тип	**Java**	**C#**
Целые числа	`int`, `long`	`int`, `long`
Числа с плавающей	`float`, `double`	`float`, `double`
Булевы значения	`boolean`	`bool`
Символ	`char`	`char`

1.4 Роль примитивных типов в управляемых средах исполнения

Управляемая среда исполнения (Managed Execution Environment) — это программный компонент или слой, который обеспечивает выполнение программного кода в контролируемой, безопасной и абстрагированной от низкоуровневых деталей среде. Она действует как посредник между скомпилированным (или интерпретируемым) кодом приложения и базовой операционной системой или аппаратным обеспечением.

1.4.1 Основные функции и компоненты среды исполнения

Виртуальная Машина (ВМ) / Интерпретатор: Ядро среды исполнения, которое выполняет промежуточный код (байт-код или Common Intermediate Language – CIL). Примеры включают Java Virtual Machine (JVM) и Common Language Runtime (CLR) для .NET. ВМ изолирует исполняемый код от конкретной аппаратной архитектуры и операционной системы.

JIT-компиляция (Just-In-Time Compilation): Для повышения производительности многие ВМ используют JIT-компиляторы. Они динамически переводят промежуточный код в машинный код непосредственно перед его выполнением. JIT-компиляторы могут применять оптимизации на основе профилирования времени выполнения, что часто позволяет достичь производительности, сопоставимой с нативно скомпилированным кодом.

Загрузчики Классов/Сборок: Эти компоненты отвечают за динамическую загрузку программных модулей (классов, библиотек, сборок) по мере их необходимости во время выполнения программы. Они также управляют разрешением зависимостей и проверкой целостности загружаемых компонентов.

Система Управления Потоками: Среда исполнения предоставляет API и механизмы для создания, управления и синхронизации потоков выполнения, обеспечивая эффективную поддержку параллельных вычислений.

Система Безопасности: ВУЯП часто включают встроенные модели безопасности (например, песочницы), которые контролируют доступ исполняемого кода к системным ресурсам, таким как файловая система, сеть или другие процессы, что особенно важно для кода, загружаемого из недоверенных источников.

Встроенные Сервисы: К ним относятся механизмы обработки исключений, рефлексия (возможность интроспекции и модификации структуры кода во время выполнения), сериализация, а также поддержка взаимодействия с нативным кодом (Native Interface).

1.4.2 Примитивные типы в управляемых средах исполнения

Примитивные типы данных в контексте управляемых сред исполнения (таких как .NET CLR и Java VM) обладают характеристиками и принципами работы, отличающимися от неуправляемых сред, например, реализованных на языках C/C++.

Абстракция и безопасность

Управляемые среды исполнения предоставляют уровень абстракции, скрывающий детали низкоуровневого представления примитивных типов данных в памяти. Это обеспечивает платформонезависимость и переносимость программного обеспечения. Кроме того, такие среды реализуют строгую проверку типов как на этапе компиляции, так и во время исполнения, что служит профилактикой ошибочного использования типов данных и возникновения уязвимостей, связанных с выделением памяти и выходом за границы допустимых областей.

Гарантированная инициализация

В отличие от неуправляемых сред, где переменные могут содержать неинициализированные значения, управляемые среды гарантируют автоматическую инициализацию переменных примитивных типов значениями по умолчанию. Это исключает возможность непредсказуемого поведения, связанного с использованием мусорных значений памяти.

Сборка мусора

В управляемых средах примитивные типы данных, как правило, не подлежат сборке мусора напрямую. Обычно они размещаются в стеке либо, если выступают в роли компонентов объектов, внутри кучи. Освобождение памяти, занятой объектами (соответственно, и их полями-примитивами), происходит посредством встроенного механизма сборки мусора, что снимает с разработчика необходимость вручную управлять памятью.

Стандартное поведение

Управляемые среды исполнения гарантируют предсказуемое и стандартизированное поведение всех операций с примитивными типами данных, в том числе арифметических. Данная особенность повышает надежность и упрощает переносимость программ между различными аппаратными и программными платформами.

Механизм упаковки и распаковки (boxing/unboxing)

Управляемые среды поддерживают механизм упаковки (*boxing*) и распаковки (*unboxing*), позволяющий преобразовывать значения примитивных типов в объекты и обратно. Это особенно актуально при взаимодействии с универсальными коллекциями и другими структурами, требующими объектного представления данных. Например, в среде C# значение типа `int` может быть преобразовано в объект (*boxing*) и обратно (*unboxing*).

Примеры реализации

- **.NET (например, на C#)**: Примитивные типы (такие как `System.Int32`) реализованы в виде структур и обеспечивают предсказуемое поведение за счёт встроенных механизмов среды.
- **Java**: Примитивные типы (`int`, `boolean`, `char`) имеют фиксированные размеры и поведение. Для поддержки упаковки и распаковки используются соответствующие классы-обёртки (`Integer`, `Boolean`, `Character`).

1.5 Несогласованность примитивных типов в управляемых языках программирования

В большинстве управляемых языков программирования (таких как Java, C#, Python) существует проблема несогласованности обработки примитивных типов данных и объектных типов в рамках единой системы типов. Эта асимметрия оказывает существенное влияние на архитектуру языка, его производительность и способность к применению принципов объектно-ориентированного программирования.

Отсутствие единообразия в системе типов

Одной из ключевых проблем является отсутствие унифицированного представления примитивных и объектных типов в системе типов управляемого языка. Это бинарное разделение приводит к концептуальной и синтаксической асимметрии. Например, примитивные типы не могут быть `null` (без дополнительных оберток), не поддерживают полиморфизм и не могут использоваться там, где требуется экземпляр класса `Object` (например, в универсальных коллекциях) без явного или неявного преобразования.

Ограничения в Объектно-Ориентированных Возможностях

Несогласованность примитивных типов создает существенные ограничения для применения объектно-ориентированных парадигм: инкапсуляция, наследование и полиморфизм.

- **Отсутствие наследования и полиморфизма:** Примитивные типы не могут наследоваться, и к ним не применимы механизмы полиморфизма через интерфейсы или абстрактные классы. Это означает, что функции, ожидающие экземпляр базового класса или интерфейса, не могут напрямую работать с примитивными значениями.

- **Проблемы с коллекциями и обобщениями (Generics):** Для включения примитивных значений в объектные коллекции (например, `ArrayList<Object>` в Java или `List<object>` в C#) требуется механизм автоматической или явной упаковки (boxing) в соответствующие объектные обертки (например, `Integer` для `int`, `Double` для `double`). Это не только нарушает прозрачность и чистоту кода, но и может приводить к неожиданным побочным эффектам или ошибкам типизации, если разработчик не учитывает процесс упаковки/распаковки.

- **Нарушение унифицированного доступа:** Объектно-ориентированный дизайн стремится к унифицированному доступу к данным и поведению через методы. Примитивные типы не обладают методами, так что разработчик должен использовать процедурные подходы или статические вспомогательные классы для выполнения операций над ними.

Производительность и накладные расходы

Примитивные типы, как было описано выше, позволяют достичь высокой производительности из-за их прямой аллокации в стеке или регистрах. Тем не менее, их несогласованность с объектной системой может приводить к значительным накладным расходам.

- **Операции упаковки и распаковки (Boxing/Unboxing):** Эти операции, интегрирующие примитивы в объектную иерархию, влекут за собой:
  - **Аллокацию памяти в куче:** Для каждого упакованного примитивного значения создается новый объект в куче, что увеличивает потребление памяти.
  - **Дополнительные циклы процессора:** Создание и инициализация объектов-оберток, а также последующее их удаление сборщиком мусора, требуют процессорного времени.
  - **Увеличение нагрузки на сборщик мусора:** Большое количество короткоживущих объектов-оберток создает дополнительную работу для сборщика мусора, потенциально приводя к задержкам в работе приложения.
- **Избыточные преобразования типов:** В сложных системах, где примитивы часто передаются между функциями, ожидающими объектные типы, и наоборот, могут возникать множественные операции упаковки и распаковки, что негативно сказывается на общей производительности системы.

Влияние на проектирование языка и его особенности

Асимметрия между примитивными и объектными типами оказывает глубокое влияние на архитектуру и проектирование самих управляемых языков, усложняя их реализацию и использование.

- **Сложность дизайна и реализации компилятора/рантайма:** Разработчикам языков приходится вводить специальные правила и исключения для обработки при-

митивных типов, которые не соответствуют общей объектной модели. Это увеличивает сложность компилятора и среды исполнения, требуя специализированных путей кода для различных типов.

- **Ограничения в расширяемости:** Механизмы расширения языка, такие как операторная перегрузка или метапрограммирование, могут быть ограничены или усложнены из-за необходимости учитывать различия между примитивами и объектами.
- **Влияние на ключевые функции языка:**
  - **Рефлексия:** Механизмы рефлексии должны предоставлять отдельные API для примитивных и объектных типов или вводить специальные обертки для работы с примитивами.
  - **Сериализация:** Унифицированная сериализация данных становится более сложной, поскольку необходимо обрабатывать как объекты, так и примитивные значения.
  - **Нулевые типы (Nullable Types):** Введение безопасных nullable-типов (например, `int?` в C#) часто требует дополнительной работы для примитивов, в то время как объектные типы по умолчанию могут быть `null`.
- **Дополнительная нагрузка на разработчика:** Разработчики, использующие управляемые языки, вынуждены постоянно учитывать различия между примитивами и объектами и помнить, когда и где следует использовать примитив, а когда — его объектную обертку, добавляет сложности в процесс разработки.

2 Постановка задачи

2.1 Контекст

В данной работе речь пойдет о высокоуровневом языке программирования с управляемой средой исполнения, поддерживающей два языка программирования — статически типизированный язык, похожий в большей степени на TypeScript, и динамически типизированный, схожий с JavaScript.

Язык поддерживает императивные, объектно-ориентированные, функциональные и шаблонные паттерны программирования и комбинирует разные семантические аспекты TypeScript, Java и Kotlin. На данный момент язык находится в активной стадии разработки.

Примитивные типы в разрабатываемом языке обладают основными характеристиками, описанными ранее — они:

- не участвуют в подтипировании (в том числе с типом `Object`)
- не могут быть компонентами юнион-типов
- не могут приниматься в generic-типах в качестве аргумента
- не имеют методов и других свойств объектов

Для обеспечения указанной функциональности язык предоставляет “Большебуквенные” аналоги примитивных типов, представляющие собой объектную обертку с соответствующими методами и правилами наследования (например, `Int` — объектный аналог примитивного типа `int`). В случаях необходимости между примитивным типом и его аналогом происходит скрытая конверсия, что усложняет семантический анализ программ. Более того, синтаксически допускаются конструкции вида `int | undefined` или `Set<int>`, которые интерпретируются компилятором как `Int | undefined` и `Set<Int>`. Несоответствие свойств `int` и `Int` приводит к непредвиденному поведению в краевых случаях.

Далее в тексте `int` и `Int` будут использоваться в качестве основного примера.

Цель работы: улучшение семантического анализа программ посредством устранения концепции примитивных типов как отдельной категории и унификации системы типов.

2.2 Задачи исследования

Для достижения поставленной цели необходимо решить следующие задачи:

1. Провести детальный анализ текущего состояния семантического анализа с целью идентификации мест, где наличие примитивных типов приводит к:
   - усложнению логики анализа
   - неоднозначности интерпретации

- необходимости специальных правил обработки (скрытые конверсии, union-типы, generics)

2. Разработать формальную спецификацию унифицированной системы типов, которая:

- исключает концепцию примитивных типов
- интегрирует их функциональность в объектную модель
- сохраняет производительность операций (арифметические, логические)
- обеспечивает корректность семантики

3. Спроектировать и реализовать модификации в подсистеме семантического анализа, обеспечивающие:

- проверку совместимости типов
- разрешение перегрузок
- вывод типов
- работу с метаданными

4. Разработать набор тестовых сценариев, охватывающих:

- generic-типы с числовыми аргументами
- union-типы
- операции с числовыми значениями
- проблемные краевые случаи

5. Оценить влияние решения на:

- сложность и точность семантического анализа
- удобство разработки
- потенциальную производительность

с использованием формальных метрик:

- уменьшение количества специальных правил
- сокращение ветвлений в коде анализатора
- результаты тестовых сценариев

2.3 Требования к решению

2.3.1 Функциональные требования

1. Унификация системы типов: Разработанное решение должно обеспечить:

- полную интеграцию всех типов (числовых, логических) в общую объектную иерархию
- участие базовых типов в механизмах подтипирования
- устранение разделения на примитивные и объектные типы

2. **Поддержка продвинутых конструкций:** Все типы должны корректно использоваться в:
   - **Union-типах:** конструкции вида `number | undefined` или `boolean | null` должны быть валидными
   - **Дженериках:** базовые типы как аргументы (`Set<number>`, `Map<string, boolean>`) без скрытых преобразований
   - **Наследовании/Подтипировании:** соблюдение общих правил для базовых типов
3. **Устранение скрытых конверсий:** Необходимо исключить:
   - автоматические механизмы упаковки (boxing)
   - автоматическую распаковку (unboxing) значений
   - неявные преобразования между разными представлениями типов
4. **Семантическая корректность:** Анализатор должен обеспечивать:
   - точное выявление типовых ошибок
   - однозначное разрешение типов в выражениях
   - предсказуемое поведение программ
5. **Сохранение синтаксиса:** Требуется:
   - сохранить существующий синтаксис (`int`, `number`, `boolean`)
   - обеспечить соответствие внутренней интерпретации унифицированной модели

2.3.2 Нефункциональные требования

1. **Производительность:**
   - отсутствие существенного замедления семантического анализа
   - оптимизация алгоритмов проверки типов
2. **Эффективность кода:**
   - отсутствие заметного снижения производительности исполняемого кода
   - сравнимая эффективность с оригинальной реализацией
3. **Качество кода анализатора:**

- повышение читаемости и поддерживаемости
- модульная структура
- упрощение логики обработки типов

4. Удобство разработки:

- интуитивно понятная модель типов
- снижение когнитивной нагрузки
- минимизация неочевидных ошибок

2.4 Ожидаемые результаты

В результате выполнения дипломной работы ожидается получить:

- **Спецификацию унифицированной системы типов** для высокоуровневого управляемого языка, включающую:
  - формальное описание типовой системы
  - правила подтипирования
  - алгоритмы проверки типов
- **Модифицированную подсистему семантического анализа** с:
  - поддержкой унифицированных типов
  - устранением специальных случаев для примитивов
  - улучшенной архитектурой
- **Комплекс тестовых примеров**, покрывающих:
  - базовые операции с типами
  - union-типы и дженерики
  - краевые случаи
  - производительность анализа
- **Оценочные метрики** по:
  - сложности семантического анализа
  - эффективности работы компилятора
  - удобству использования языка

3 Обзор существующих решений

Интеграция примитивных типов в высокоуровневую систему типов и их влияние на семантический анализ

3.1 Явное разделение с автоматическим преобразованием

- **Суть:** Примитивные типы (‘int‘, ‘float‘, ‘boolean‘) и объектные типы (‘Integer‘, ‘Float‘, ‘Boolean‘) существуют параллельно и явно различимы в системе типов языка. Компилятор автоматически вставляет преобразования (боксинг - примитив -> объектная обертка, анбоксинг - объектная обертка -> примитив) там, где контекст требует типа другого вида (например, передача ‘int‘ в метод, ожидающий ‘Object‘, или использование ‘Integer‘ в арифметической операции).
- **Влияние на систему типов:**
  - Явный дуализм типов
  - Примитивы не являются частью объектной иерархии
  - Необходимость учёта примитивных типов в правилах подтипирования
- **Влияние на семантический анализ: Значительно усложняет анализ**
  - Отслеживание контекстов, требующих преобразований
  - Разрешать перегрузки методов с примитивными и объектными параметрами (иногда приводя к неочевидному выбору)
  - Обрабатывать потенциальные `NullPointerException` при анбоксинге `null`
  - Учитывать различия в семантике (например, `==` для примитивов vs. для объектов)
  - Иметь отдельные ветви кода для проверки типов примитивов и объектов
- **Преимущества:** Позволяет использовать примитивы для производительности и объектные обертки там, где нужна полиморфность (коллекции). Понятна разработчикам низкого уровня.
- **Недостатки:** Сложная система типов и семантический анализ, риск ошибок ‘NullPointerEx из-за неявного анбоксинга, потенциальные накладные расходы на преобразования, концептуальный разрыв для разработчика.
- **Примеры:** Java (классический пример), ранние версии C#

```java
Integer x = 10;   // Автобоксинг
int y = x;        // Автоанбоксинг
if (y == x) {     // Сравнение с автоанбоксингом
    System.out.println("Equal");
}
```

Пример в Java

3.2 Единая Иерархия Типов с Значимыми Типами

– **Суть:** Примитивные типы реализованы как **значимые типы (value types)**, которые являются частью единой объектной иерархии типов (например, наследуются от базового класса `ValueType`, который сам наследуется от `Object`). Все типы (и ссылочные, и значимые) формально являются подтипами Object. Однако, между ссылочными (классы) и значимыми (структуры, примитивы) типами сохраняется фундаментальное различие в семантике: передача по ссылке и передача по значению (копированию), размещение в куче и в стеке/встроенно.

– **Влияние на систему типов:** Формально единая иерархия, но с глубоким внутренним разделением. Примитивы/значимые типы могут реализовывать интерфейсы. Возможно ограниченное наследование для значимых типов (или его отсутствие).

– **Влияние на семантический анализ: Усложнен.** Анализатор должен:

  – Различать ссылочные и значимые типы на протяжении всего анализа.

  – Учитывать семантику копирования при присваивании и передаче в методы для значимых типов.

  – Обрабатывать боксинг/анбоксинг (упаковку/распаковку) при необходимости преобразования значимого типа в ссылочный (`object`, интерфейс) и обратно, со всеми вытекающими последствиями (накладные расходы, `Null`?).

  – Учитывать различия в поведении операторов (например, `==` по умолчанию для значимых типов сравнивает значения, а для ссылочных - ссылки).

– **Преимущества:** Более единообразная модель типов, чем в Java. Значимые типы позволяют создавать эффективные пользовательские структуры данных. Возможность полиморфизма через интерфейсы.

– **Недостатки:** Сохраняется концептуальная сложность разделения ref/value. Семантический анализ все еще должен обрабатывать два разных вида типов и преобразования между ними. Риск неочевидных накладных расходов на упаковку.

– **Примеры:** C# (структуры `struct`, примитивы как псевдонимы для системных структур типа `System.Int32`), Swift (value semantics для структур и перечислений).

3.3 Примитивы как оптимизация компилятора (Инлайнинг)

– **Суть:** Система типов языка оперирует только высокоуровневыми объектными типами. Примитивные типы абстрагированы на уровне семантики языка. Компилятор на поздних стадиях (после семантического анализа) агрессивно оптимизирует использование этих объектов:

  – заменяет их на низкоуровневые примитивные значения,

  – подставляет реализацию их методов напрямую в код,

  – устраняет накладные расходы на вызовы методов и выделение памяти.

  Примитив используется только как реализационная деталь оптимизации, невидимая для семантики языка.

– **Влияние на систему типов: Единая объектная модель.** Примитивы представлены как специальные неизменяемые объектные типы (часто `final` классы или `inline` классы). Для системы типов и разработчика они:

  – выглядят и ведут себя как обычные объекты,

  – имеют методы,

  – могут быть упакованы в общие интерфейсы.

– **Влияние на семантический Анализ: Существенно упрощен.** Анализатор работает исключительно с объектными типами. Нет необходимости:

  – Различать примитивы и объекты при проверке типов.

  – Обрабатывать правила боксинга/анбоксинга (их нет на уровне семантики).

  – Беспокоиться о `NullPointerException` для самих объектных аналогов примитивов.

– **Преимущества:**

  – Максимально простая и единообразная система типов,

  – Упрощенный семантический анализ для высокоуровневых конструкций,

  – Сохранение производительности примитивов через оптимизацию,

  – Высокая абстракция для разработчика.

– **Недостатки:**

  – Сложность реализации оптимизирующего компилятора,

  – Необходимость четких правил для компилятора (например, часто требуется `final`/неизм

  – Возможные ограничения:

    * массивы примитивных объектов могут быть менее эффективны,

    * работа с рефлексией может показывать обертку.

- **Примеры:**
  - Kotlin (`inline` классы для представления примитивоподобных типов),
  - Scala (классы-значения `AnyVal` и их подклассы для `Int`, `Double` и т.д. - хотя в Scala есть и примитивы JVM, `AnyVal` абстрагирует их на уровне семантики Scala).

3.4 Полное Удаление Примитивных Типов

- **Суть:**
  - В системе типов языка отсутствует само понятие "примитивный тип"
  - Все данные (целые числа, числа с плавающей точкой, булевы значения, символы) являются полноценными объектами
  - Операции над ними реализованы сключительно как методы этих объектов
  - Любая низкоуровневая оптимизация(представление в виде примитивных значений процессора или inline-подстановка операций) является исключительной задачей компилятора или среды выполнения:
    * Происходит после этапа семантического анализа
    * Не влияет на правила языка и работу семантического анализатора
- **Влияние на Систему Типов:**
  - **Абсолютно единая объектная иерархия**
  - Все типы являются объектными
  - Нет дуализма или разделения на ref/value на уровне абстракции языка
- **Влияние на Семантический Анализ:**
  - **Максимально упрощен и унифицирован**
  - Анализатор:
    * Работает только с объектными типами и их методами
    * Проверяет типы и разрешает перегрузки методов по единым правилам
    * Обрабатывает операции как обычные вызовы методов (`a.plus(b)`, `x.equals(y)`)
    * Не содержит никакого кода, специфичного для обработки примитивов
- **Преимущества:**
  - Наивысшая степень абстракции и согласованности
  - Простота семантического анализа

  - Концептуальная чистота
  - Устранение целых классов ошибок, связанных с примитивами
- **Недостатки:**
  - Высокие требования к оптимизирующему компилятору/рантайму
  - Потенциальные сложности с низкоуровневым взаимодействием
  - Исторически возможная неестественность представления операций как методов
- **Примеры:**
  - Smalltalk
  - Ruby (и объекты, и числа имеют методы)

4 Исследование и построение решения задачи

4.1 Архитектурное решение

В работе реализована стратегия абстрагирования примитивных типов посредством их представления в виде объектов высокоуровневой системы типов, где примитивные операции (арифметика, сравнения) выполняются непосредственно над объектным аналогом примитивного типа. Физическое представление примитивов и низкоуровневые оптимизации генерируются компилятором после этапа семантического анализа, не влияя на корректность и согласованность правил типизации.

4.2 Обоснование выбора стратегии

На основании проведённого обзора существующих решений выбрана стратегия «Примитивы как оптимизация компилятора» по следующим причинам:

1. **Упрощение семантического анализа**
   - Анализатор работает только с объектными типами (например, `Int` как класс-обёртка)
   - Устранение дуализма типов (как в Java) и проблем упаковки (как в C#)
2. **Совместимость с целями языка**
   - Чистая объектная модель критична для семантической согласованности высокоуровневого управляемого языка.
3. **Практическая эффективность**
   - Оптимизации (инлайнинг методов, анбоксинг) выполняются на последних этапах и не нарушают семантику проверки типов.

4.3 Описание решения

- Все примитивоподобные типы становятся экземплярами Object
- Замена на низкоуровневые примитивы выполняется только на этапе оптимизации компиляции
- Именение семантики массивов во время исполнения во избежание неэффективной работы с ними

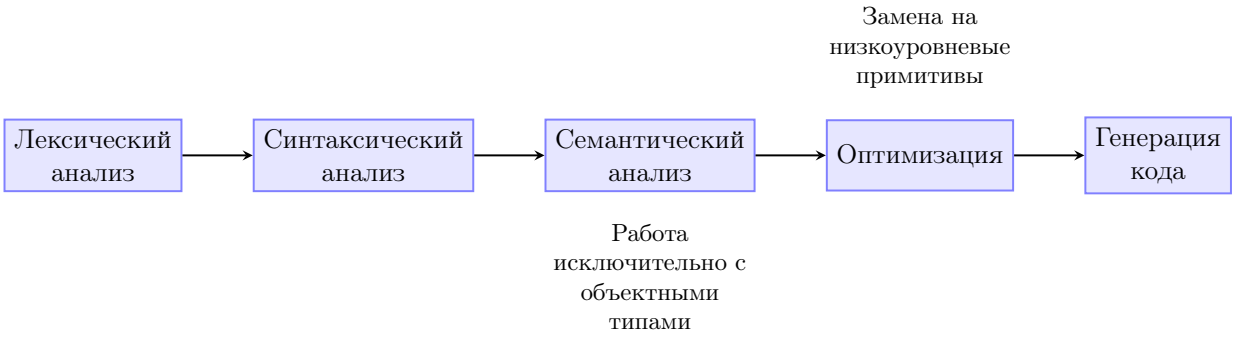

Рис. 1: Этапы работы компилятора с выделением семантического анализа

Новая унифицированная система типов предусматривает:

1. Каждый примитивный тип полностью эквивалентен своей объектной обёртке, являющейся подклассом Object (включая специальные типы: void/Void, undefined/Undefined, null/Null)
   - Арифметические операции выполняются непосредственно над объектами без распаковки
   - Тип объектный примитивоподобный тип становится полноценным участником union-типов
   - Операторы `==` и `===` реализуют сравнение по значению (аналогично поведению для String)

4.4 Изменения в семантике языка

4.4.1 Эквивалентные сценарии

Для наглядности далее в тексте в качестве базового примера будут использоваться типы: int (примитивный) → Int (объектный аналог)

Примитивы	Объектные аналоги
`let x = 1.0`	`let XX = new Nummber(1.0)`
`x++`	`XX++ [XX = new Number(XX.value + 1), неявный боксинг]`
`x == 0`	`XX == 0 [XX.unboxed() == 0, неявный анбоксинг]`
`x === 0`	`XX === 0`
`function f(XX: Number) {}; f(x)` `[неявный боксинг]`	`function f(x: number) {}; f(XX) [неявный анбоксинг]`
`function g<T>(t: T) {}; g(x) [неявный боксинг]`	`function g<T>(t: T) {}; g(XX)`
`let u: number | null = 0 [неявная замена на Number]`	`let UU: Number | null = 0`
`g<number>(0) [неявная замена на Number]`	`g<Number>(0)`
`typeof(x) == "number"`	`typeof(XX) == "Number"`

Рис. 2: Красными комментариями помечены места, в которых происходили неявные упаковки и распаковки до удаления примитивных типов

4.4.2 Новые допустимые конструкции

```
xx.toStrint() // ошибка компиляции до изменений, будет работать после
xx instanceof Object // ошибка компиляции до изменений, будет работать после
```

Пример в Java

4.5 План реализации

1. **Модификация системы типов**: Убрать примитивные типы на этапе проверки корректности типов в арифметических операциях и преобразованиях типов

2. **Обработка констант**: Изменить представление литералов на этапе свёртки констант на объектные типы

3. **Оптимизация перед кодогенерацией**: Имплементировать модуль-оптимизацию, где объектные типы будут по возможности заменяться примитивными

5 Описание практической части

Компилятор рассматриваемого языка в основном написан на языке программирования C++.

Для наглядности далее в тексте в качестве базового примера будут использоваться типы: int (примитивный) → Int (объектный аналог)

5.1 Арифметические операции

На данном этапе работы было необходимо модифицировать проверку корректности типов в арифметческих операциях:

`+` , `-` , `+=` , `-=`

`++` , `--`

`*` , `/` , `%` , `*=`, `/=`, `%=`

`<<` , `>>` , `<<=` , `>>=`, `>>>`, `>>>=`

`|` , `|=` , `&` , `&=`, `^`, `^=`, `||` , `&&`

`<` , `<=` , `>`, `>=`, `==`, `===`, `!`

В существующей реализации почти в каждой функции проверки производилось безусловное приведение типов операторов к примитивным, поэтому первым этапом работы было удаление или изменение кода, который приводил бы к замене объектных типов на их примитивные аналоги. Далее, основная функция, которая была нужна почти для всех вышеперечисленных операций(исключая логические и унарные) - функция, вычисляющая тип результата бинарного выражения. Ее поведение основывается на иерархии численных типов, оговоренной в спецификации. Следующий этап заключался в очистке кода от попыток свернуть константы в функциях проверки, чтобы вынести все константные вычисления в отдельный модуль.

5.2 Преобразования типов

Для реализации проверки легальности преобразования объектных примитивоподобных типов можно было переиспользовать алгоритм, использовавшийся для проверки преобразований примитивных типов, добавив дополнительные проверки на отношения объектов (Union и Enum). Структура функции для проверки корректности преобразования представлена на 3

```
bool IsLegalBoxedPrimitiveConversion(Type *target, Type *source)
{
    if (target == nullptr || source == nullptr) {
        return false;
    }

    if (target->IsUnionType() && source->IsObjectType()) {
        // ...Логика выбора примитивного юнион типа...

        // Переиспользование проверки для примитивных типов
        bool res = Result(IsAssignableTo(sourceUnboxedType, target));
        return res;
    }
    // ...Проверки, что операнды являются обернутыми примитивами...

    // распаковка операндов
    Type *targetUnboxedType = UnboxType(target);
    Type *sourceUnboxedType = UnboxType(source);

    if (source->IsIntEnumType()) {
        // ...Логика при сравнении enum типов...
    }
    // ...Проверки валидности распакованных типов...

    if (targetUnboxedType == nullptr || sourceUnboxedType == nullptr) {
        return false;
    }
    if (!targetUnboxedType->IsPrimitiveType() || !sourceUnboxedType->IsPrimitiveType()) {
        return false;
    }

    // Переиспользование проверки для примитивных типов
    bool res = this->Result(this->IsAssignableTo(sourceUnboxedType, targetUnboxedType));
    return res;
}
```

Рис. 3: Структура функции для проверки корректности преобразования примитивоподобных объектов

5.3 Свёртка констант

Один из основных этапов работы - это имплементация модуля компиляции для сверки констант в начало стадии семантического анализа. Свёртка констант - вычисление константных выражений с последующей заменой выражений на результаты, выполняемое на отдельной стадии компиляции. Это оптимизация, которая:

– Ускоряет выполнение программы (избегает вычислений в runtime)

– Уменьшает размер генерируемого кода

– Позволяет обнаружить ошибки на этапе компиляции

Исплементированный модуль позволяет рекурсивно обойти const и readonly декларации и заменить константные выражения результатом их вычисления.

5.3.1 Основной алгоритм

– **Обход AST** - рекурсивных обход синтаксического дерева программы

- **Идентификация константных выражений** - проверка, можно ли вычислить выражение на этапе компиляции (является ли выражение константным)
- **Вычисление констант** - выполнение операций над константами
- **Замена выражений** - подстановка вычисленных значений вместо исходных выражений

5.3.2 Поддерживаемые типы и операции

Типы данных

- Числовые литералы (int, float, double)
- Символьные литералы (char)
- Булевы значения (true/false)
- Строковые литералы
- Enum значения

Операции

- Арифметические +, -, *, /, %
- Битовые &, |, ^, ~, <<, >>, >>>
- Логические &&, ||, !
- Сравнения ==, !=, <, >, <=, >=
- Унарные +, -, ~

5.3.3 Детали реализации

Типизация и преобразование типов

- Используется система рангов типов TypeRank для определения приоритета преобразований
- Реализованы безопасные преобразования между типами с проверкой диапазонов
- Обработка деления на ноль

```
enum class TypeRank {
    INT8,
    INT16,
    INT32,
    INT64,
    FLOAT,
```

```
    DOUBLE,
    CHAR
};
```

Листинг 1: Перечисление рангов типов

5.3.4 Основные функции

```
static TypeRank GetTypeRank(const ir::Literal* lit) {
    if (lit->IsCharLiteral()) {
        return TypeRank::CHAR;
    }

    auto num = lit->AsNumberLiteral()->Number();
    if (num.IsByte()) return TypeRank::INT8;
    if (num.IsShort()) return TypeRank::INT16;
    if (num.IsInt()) return TypeRank::INT32;
    if (num.IsLong()) return TypeRank::INT64;
    if (num.IsFloat()) return TypeRank::FLOAT;
    if (num.IsDouble()) return TypeRank::DOUBLE;

    ES2PANDA_UNREACHABLE();
}
```

Листинг 2: GetTypeRank

Получение значения литерала

```
template <typename TargetType>
static TargetType GetVal(const ir::Literal* node) {
    // Обработка булевых литералов
    if constexpr (std::is_same_v<TargetType, bool>) {
        return node->AsBooleanLiteral()->Value();
    }

    // Обработка символьных литералов
    if constexpr (std::is_same_v<TargetType, char16_t>) {
        return node->AsCharLiteral()->Char();
    }

    auto numNode = node->AsNumberLiteral();

    // Обработка числовых литералов разных типов
    if constexpr (std::is_same_v<TargetType, int8_t>) {
        return numNode->Number().GetByte();
    }
```

```
    if constexpr (std::is_same_v<TargetType, int16_t>) {
        return numNode->Number().GetShort();
    }
    // ...
    if constexpr (std::is_same_v<TargetType, double>) {
        return numNode->Number().GetDouble();
    }
}
```

Листинг 3: GetVal

Преобразование значений между типами

```
template <typename To>
static To CastValTo(const ir::Literal* lit) {
    // Обработка булевых литералов
    if (lit->IsBooleanLiteral()) {
        return static_cast<To>(GetVal<bool>(lit));
    }

    // Определение ранга типа и преобразование
    auto rank = GetTypeRank(lit);
    switch (rank) {
        case TypeRank::DOUBLE:
            return static_cast<To>(GetVal<double>(lit));
        case TypeRank::FLOAT:
            return static_cast<To>(GetVal<float>(lit));
        // ...
        case TypeRank::CHAR:
            return static_cast<To>(GetVal<char16_t>(lit));
    }

    ES2PANDA_UNREACHABLE();
}
```

Листинг 4: CastValTo

Обработка ошибок

– Генерация диагностических сообщений для недопустимых операций

– Реализованы безопасные преобразования между типами с проверкой диапазонов

– Отдельные функции преобразований между целыми и вещественными типами

Оптимизации

– Использование битовых операций для безопасной работы с целыми числами

– Специальная обработка строковых конкатенаций

– Оптимизация шаблонных литералов

5.4 Оптимизация

Реализованная оптимизация представляет собой систему преобразования типов, которая рекурсивно анализирует и модифицирует абстрактное синтаксическое дерево (AST) для выполнения операций nboxing (распаковки) и boxing (упаковки) типов.

Архитектура оптимизации

5.4.1 Класс UnboxVisitor

Рекурсивный анализ дерева проводится с помощью специального класса `UnboxVisitor`, реализующего методы `VisitX` для различных типов узлов AST. Его задачи:

1. Обойти AST и найти места, где можно заменить boxed-типы на примитивы
2. Вставить явные преобразования (например, вызовы `unboxed()`, `valueOf()`, intrinsic-функции)
3. Обновить соответствующие типы в выражениях, объявлениях и сигнатурах
4. Заново запустить функцию валидации ноды

5.4.2 Обрабатываемые узлы AST

Примеры обрабатываемых узлов:

- `VisitCallExpression` — вызовы функций
- `VisitBinaryExpression` — бинарные операции (+, ==, && и т. д.)
- `VisitMemberExpression` — доступ к полям/методам (`obj.field`, `arr[index]`)
- `VisitReturnStatement` — возвращаемые значения
- `VisitVariableDeclarator` — объявления переменных

5.4.3 Механизм работы Visitor'ов

Каждый узел AST имеет метод `Accept(visitor)`. При вызове `astNode->Accept(visitor)` управление передаётся в соответствующий метод `VisitX` у UnboxVisitor.

5.4.4 Пример обработки CallExpression

```
1 void VisitCallExpression(ir::CallExpression *call) override {
2     // 1. Обновление типов аргументов
3     for (size_t i = 0; i < call->Arguments().size(); i++) {
4         auto *arg = call->Arguments()[i];
```

```
        auto *expectedType =
    call->Signature()->Params()[i]->TsType();
        call->Arguments()[i] = AdjustType(uctx_, arg, expectedType);
    }

    // 2. Обновление возвращаемого типа
    if (call->Signature()->ReturnType()->IsETSPrimitiveType()) {
        call->SetTsType(call->Signature()->ReturnType());
    }
}
```

Листинг 5: Обработка вызовов функций

5.4.5 Логика преобразования типов

Решение о том, нужно ли вставлять boxing, unboxing или конверсию примитивов, принимается с помощью метода AdjustType:

```
static ir::Expression *AdjustType(UnboxContext *uctx,
                                  ir::Expression *expr,
                                  checker::Type *expectedType) {
    // Если выражение - примитив, а ожидается объект → boxing
    if (expr->TsType()->IsETSPrimitiveType()
        && expectedType->IsETSObjectType()) {
        return InsertBoxing(uctx, expr);
    }
    // Если выражение - boxedобъект-, а нужен примитив → unboxing
    if (TypeIsBoxedPrimitive(expr->TsType())
        && expectedType->IsETSPrimitiveType()) {
        return InsertUnboxing(uctx, expr);
    }
    // Если оба примитива, но разных типов → конверсия
    if (expr->TsType()->IsETSPrimitiveType()
        && expectedType->IsETSPrimitiveType()) {
        return InsertPrimitiveConversion(uctx, expr, expectedType);
    }
    return expr;
}
```

Листинг 6: Метод AdjustType

5.4.6 Пример преобразования

Исходный код

```
let x: Int = 10;  // Int - объектный тип
let y: int = x + 5;   // int - примитив
```

Преобразование AST

```
BinaryExpression(
    left: Identifier("x", type=Int),
    op: +,
    right: NumberLiteral(5, type=int)
)
```

Листинг 7: AST до оптимизации

```
    BinaryExpression(
        left: CallExpression(
            callee: MemberExpression(
                object: Identifier("x", type=Int),
                property: "unboxed"
            ),
            type=int
        ),
        op: +,
        right: NumberLiteral(5, type=int)
    )
```

Листинг 8: AST после оптимизации

6 Заключение

Здесь надо перечислить все результаты, полученные в ходе работы. Из текста должно быть понятно, в какой мере решена поставленная задача.

Список литературы

[1] *Mott-Smith, H.* The theory of collectors in gaseous discharges / H. Mott-Smith, I. Langmuir // *Phys. Rev.* — 1926. — Vol. 28.

[2] *Морз, Р.* Бесстолкновительный PIC-метод / Р. Морз // Вычислительные методы в физике плазмы / Ed. by Б. Олдера, С. Фернбаха, М. Ротенберга. — М.: Мир, 1974.

[3] *Киселёв, А. А.* Численное моделирование захвата ионов бесстолкновительной плазмы электрическим полем поглощающей сферы / А. А. Киселёв, Долгоносов М. С., Красовский В. Л. // Девятая ежегодная конференция «Физика плазмы в Солнечной системе». — 2014.